Aspect-Oriented Multi-Client Chat Application

Duygu Ceylan, Gizem Gürcüoğlu, Sare G. Sevil

Bilkent University,
Department of Computer Engineering.
Ankara, Turkey

Outline

- Introduction
- OO Design
- AO solution (*using AspectJ*)
- Alternative implementation: JBoss
- Related Work
- Discussion

Distributed Systems

- Peer-to-peer
- Client-Server
- N-tier
- …

Client-Server Model

- Widely used
- Many different application areas:
  - Business applications
  - HTTP, SMTP, DNS, etc.
- Two main systems:
  - Client (s)
  - Server

Client-Server Model

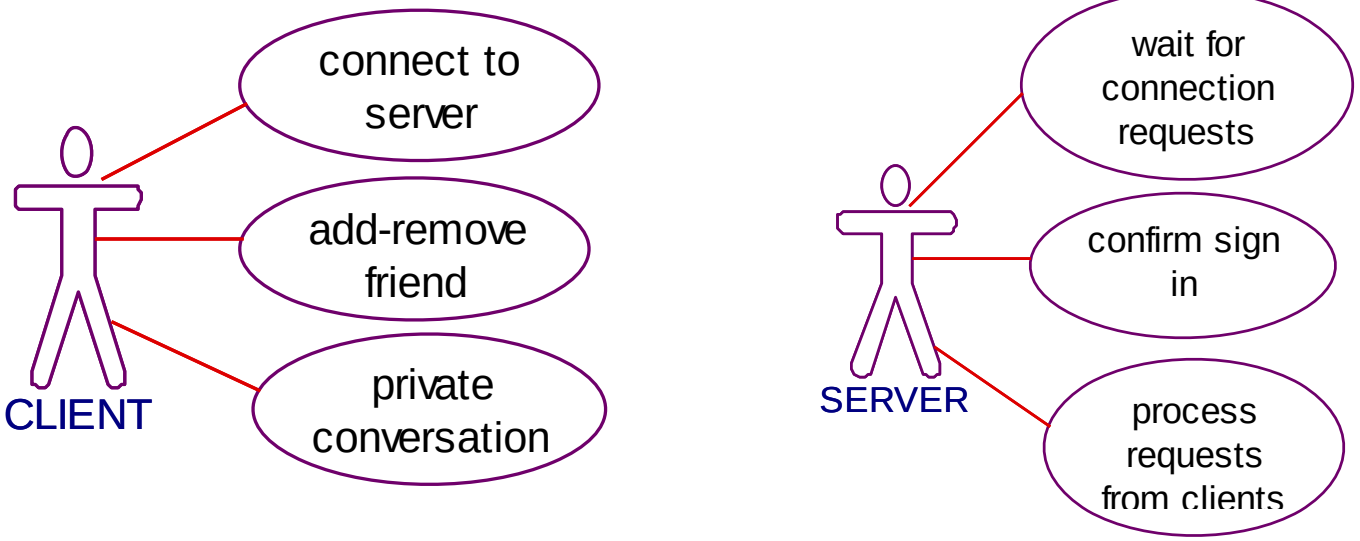
One implementation: Chat Application

- Communication between users
- Two types:
  - ChatRooms
  - Instant Messengers (IM)

Our focus: Design and implementation details of Multi-Client Chat using AOP.

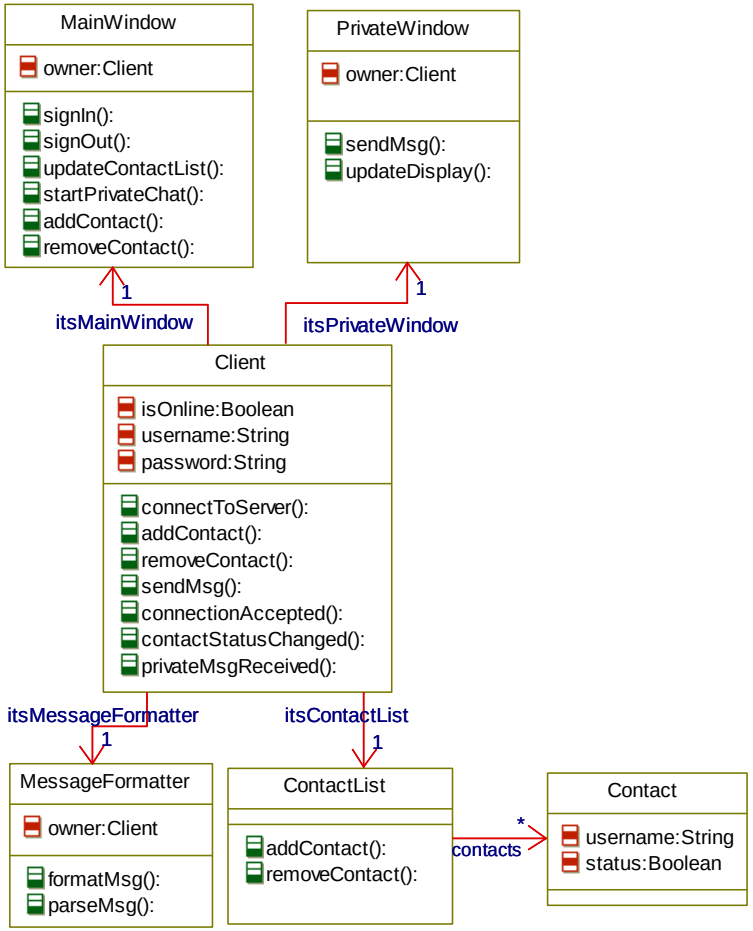
Chat Application - OO Design

- Two main components:
  - Client
  - Server



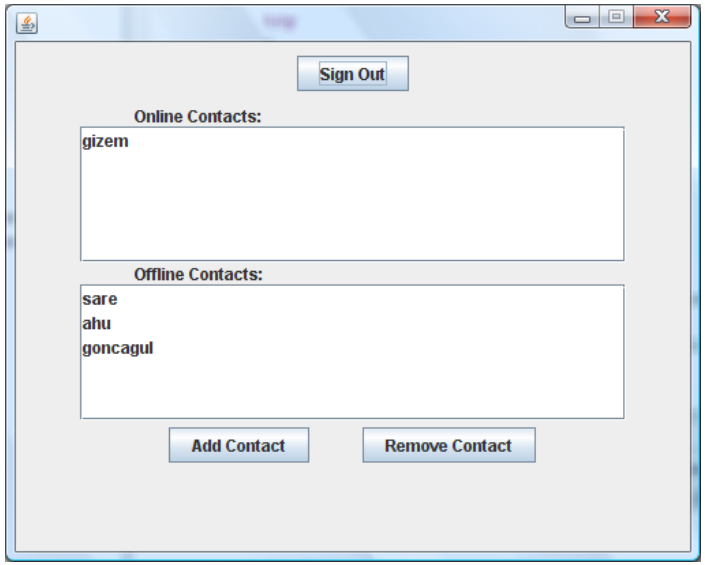
Chat Application - OO Design

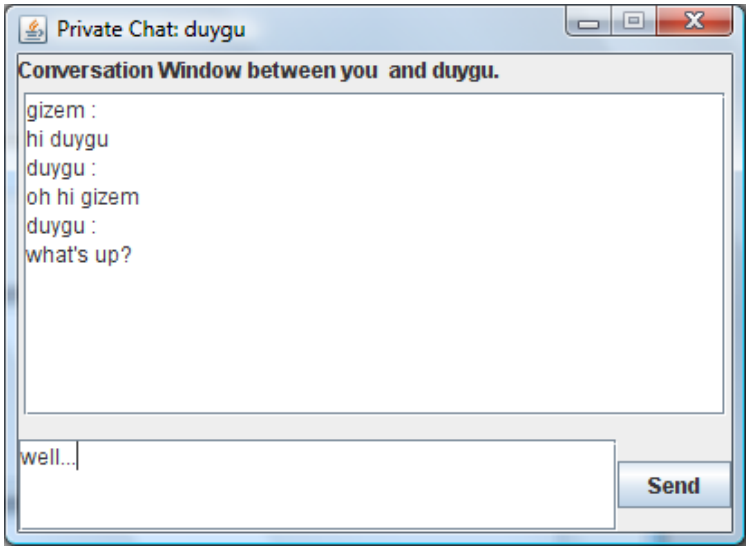
Client:



Chat Application - OO Design

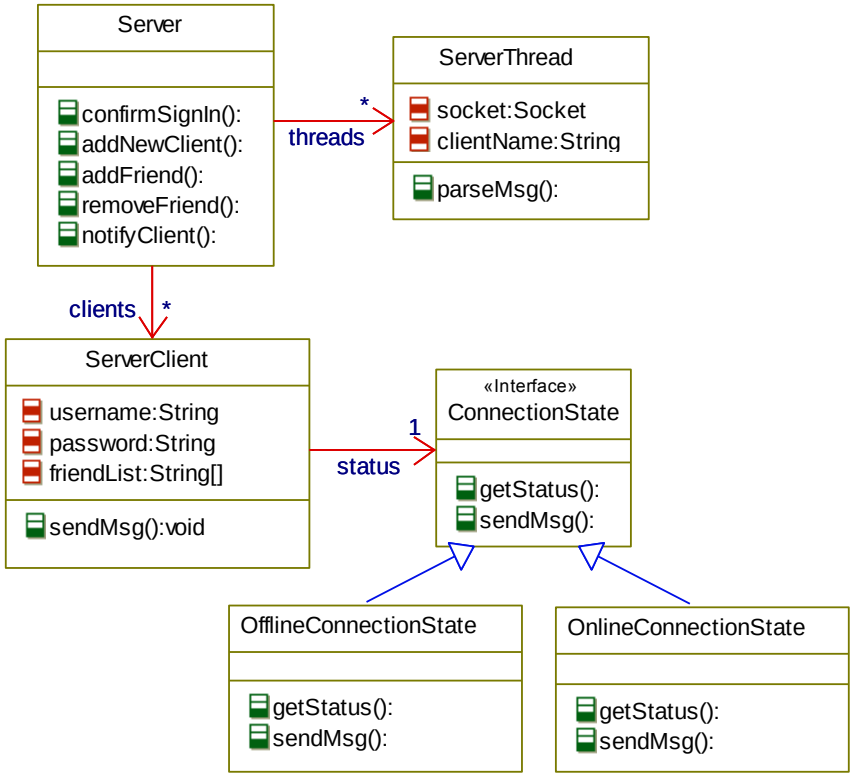
- Client (continued)





Chat Application - OO Design

- Server:



Chat Application - OO Design

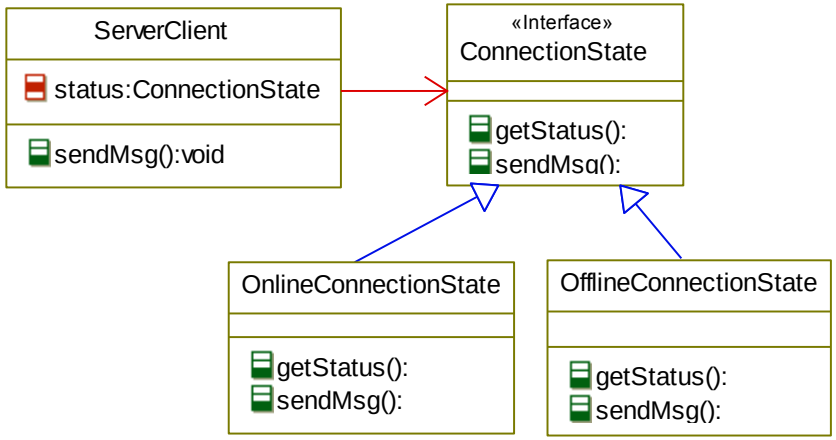
Design patterns used:

- Observer Pattern
- State Pattern

Chat App - OO Design

State Pattern:

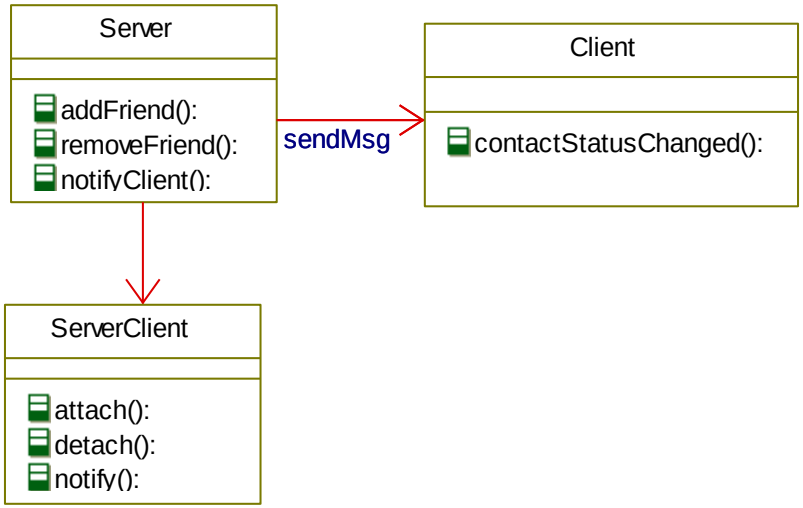
- Clients can be in one of different states.
  - Currently either *online* or *offline*
  - *Busy*, *away*, …



Chat App - OO Design

Observer Pattern

- *Clients change status*
  - Other clients need to be notified, *contact lists need to be updated*



Chat App - AOP Design

- Four Aspects:

(Production Aspects)

1. Message Send Failure Handling
2. Audio Notification

(Development Aspects)

3. Profiling system requirements
4. Controlling Client class access

AOP Design – Production Aspects

- Affects the production phase
- Adds functionality to the software
- Facilitates implementation
- Separates cross-cutting concerns

AOP Design - Message Send Failure Handling

- Private chat windows control conversations between client pairs.
- Clients can change states at any time.
- Observer pattern: notifies and updates *contact lists* of clients.
- *We need to inform 'chatting' client. How?*

AOP Design - Message Send Failure Handling

```
pointcut msgSendFailedNotification(Server s, String
                username, String receiver, String msg)
     : call(* Server.sendPrivateMsg(..)) &&
       target (s) && args(username, receiver, msg);

after(Server s, String username, String receiver, String msg)
throwing(): msgSendFailedNotification(s, username, receiver, msg)
{
  System.out.println("exception aspect");
  DataOutputStream dout = s.getDout(username);

  try
  {
     dout.writeUTF(header + receiver + separator
                                     + notification);
}
```

AOP Design – Audio Notification

- Users need to be informed about changes that occur in the application: *status change of contacts, incoming messages…*
- Having notification sounds would provide a friendlier user interface.
- *But: having too many notifications sounds would result in scattered code.*

Audio Notification – “User Signed In”

```
pointcut signedIn() : execution (void
    client.connectionAccepted(..));

void around() : signedIn()
{
   New
     AudioWavPlayer(signedInAlertFile).start()
     ;
   proceed();
}
```

Audio Notification – "Message Received"

```
pointcut receivedMsg() :
call(void Client.privateMessageReceived(..));

void around() : receivedMsg()
{
    New AudioWavPlayer(receivedMessageAlertFile).start();
    proceed();
}
```

AOP Design – Development Aspects

- Intended for usage in development stages.
- Facilitate: testing, debugging, defining enforcement policies and performance tuning work.
- Modularized functionality
  - Thus, can easily be removed from system.

AOP Design - Profiling System Requirements

- Keeping track of server work load is important:
  - When people log-in
  - When people log-out
  - How long on average they stay online
  - How many people do they talk to
  - (if system supports file transfer) when and how much transfer is done
- These requirements are scattered in the code

AOP Design - Profiling System Requirements

```
pointcut startServer() : execution(*
                              Server.listen(..));
before() : startServer() {
    try {
  Calendar cal = Calendar.getInstance();
  SimpleDateFormat sdf = new

    SimpleDateFormat("yyyyMMdd_hhmm");
  File f = new File("bin\\log" + sdf.format(cal.getTime())
                                                    +
    ".txt");
  wr = new BufferedWriter(new FileWriter(f));
  signInTimes = new Hashtable<String,String>();
  }
  catch(IOException ex)  { ex.printStackTrace();  }
}
```

AOP Design - Profiling System Requirements

```
pointcut stopServer() : execution(*
    Server.stopServer(..));
after(): stopServer()
{
  try {
    wr.close();
  }
  catch(IOException ex) {}
}
```

AOP Design - Profiling System Requirements

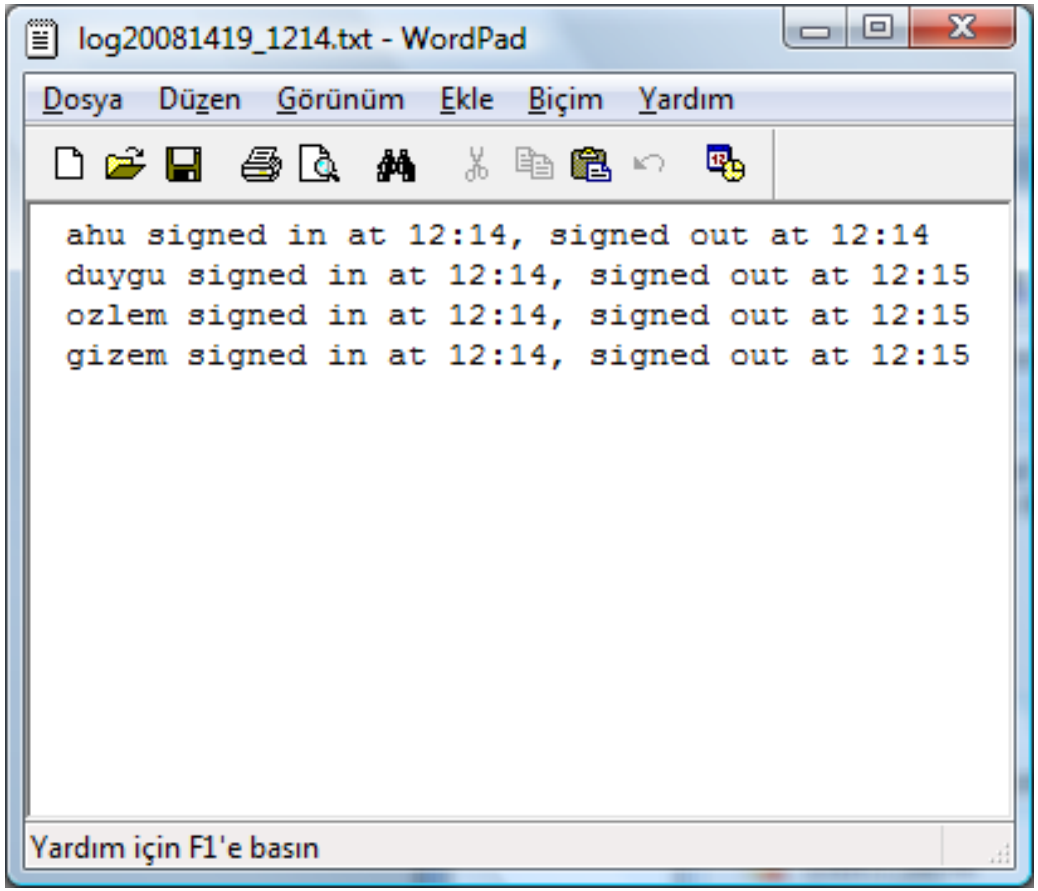
```
pointcut signIn(String username, Socket socket) :
  args(username, socket) &&
    execution(* Server.clientConnected(..));
after(String username, Socket socket)
    returning() : signIn(username, socket) {
  Calendar cal = Calendar.getInstance();
  SimpleDateFormat sdf = new
  SimpleDateFormat("hh:mm");
  signInTimes.put(username,sdf.format(

    cal.getTime()));
}
```

AOP Design - Profiling System Requirements

```
pointcut signOut (String username, Socket socket) :
    args (username, socket) &&
   execution(* Server.removeConnection(..));
after(String username, Socket socket)
     returning() : signOut(username, socket) {
  Calendar cal = Calendar.getInstance();
  SimpleDateFormat sdf = new SimpleDateFormat("hh:mm");
  try {
    wr.write(username + "\t\tsigned in at " +
      signInTimes.get(username) + ", signed
      out at " + sdf.format(cal.getTime()) +
      "\n");
  }  catch(IOException ex)  {}
}
```

AOP Design - Profiling System Requirements



AOP Design - Controlling Client Class Access

In our implementation:

- Extensive Client class
- Server class needs to store some Client related info.

- ServerClient class: contains server related Client info

- *Server class should not access Client class directly.*

AOP Design - Controlling Client Class Access

```
declare error : call(Client.new(..))&&
   within(Server) :
   "Client cannot be created in Server.";
```

Chat Application - JBoss

We have used AspectJ for AOP.

- An alternative would be JBoss

The two frameworks are quite different:

- Pointcuts in JBoss:
  using XML or Java annotations
- Aspects are encapsulated in java classes:
  that implement the Interceptor interface.
- Compiling and deploying: more difficult.

Chat Application - JBoss

```
public class AlertingInterceptor implements Interceptor {
    public String getName() { return AlertingInterceptor; }
  public InvocationResponse invoke(Invocation invocation)
                                                  throws Throwable {
    new AudioWavPlayer (receivedMessageAlertFile).start();
     return invocation.invokeNext();
   }
}
<bind pointcut="public void Client->
privateMessageReceived(String name,
String msg)">
<interceptor class="AlertingInterceptor"/>
</bind>
```

Related Work

S. Subotic et. al.

"Aspect-Oriented Programming for a Distributed Framework"

SACJ, 2006.

P. Falcarin et. al.

"Remote Trust with Aspect-Oriented Programming"

AINA, 2006.

Conclusion

Thank you for listening.

Questions?

References

- S. Subotic, J. Bishop, and S. Gruner, "Aspect-Oriented Programming for a Distributed Framework", in the *Proceedings of SACJ*, 2006.
- P. Falcarin, R. Scandariato, and M. Baldi, "Remote Trust with Aspect-Oriented Programming", in the *Proceedings of the 20th International Conference on Advanced Information Networking and Applications*, 2006.
- http://en.wikipedia.org/wiki/Client-server